VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Peer-to-Peer Instant Messenger

2010 **Daniel Robenek**

Zadání bakalářské práce

Peer-to-Peer Instant Messenger

Navrhněte a pokuste se naimplementovat instant messenger nezávislý na jakémkoliv serveru. Výsledný program by měl mít uživatelsky přívětivé gui se snadným ovládáním. Měl by implementovat většinu těchto vlastností:

1. GUI rozhraní.
2. Nezávislost na serveru.
3. Vytváření skupin.
4. Možnost posílání souboru.
5. Oznámení při příchozí zprávě / oznámení přihlášení uživatelů.
6. Historie zpráv.
7. Přenos dat pomocí paketu.
8. Šifrování posílaných zpráv.

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 7. května 2010

_____

vlastnoruční podpis autora

Poděkování

Tímto bych rád poděkoval vedoucímu práce Ing. Davidu Seidlovi za cenné rady, připomínky a metodické vedení práce.

Abstrakt

Tato bakalářská práce klade za cíl objasnit návrh implementace komunikačního klienta v peer-to-peer modelu sítě. Postupně budou rozebrány problémy vzniklé díky tomuto modelu, jejich možná řešení a popis řešení nejvhodnějšího. Dalším krokem bude návrh implementace takovéhoto komunikačního klienta, včetně popisu všech jeho nejdůležitějších součástí. Jako další cíl je samotná implementace, jež má obsahovat co nejvíce vlastností daných v zadání.

Abstract

The bachelor thesis aims to explain the design of implementation of the communication client peer-to-peer network model. Gradually, they discussed the problems caused by this model, possible solutions and the most appropriate description of the solution. The next step is the design of implementation, including a description of all its major components. As a further objective is the implementation itself, which is to include as many features, given the assignment.

Klíčová slova

Komunikační klient, peer-to-peer síťový model, java, broadcast, distribuce dat

Keywords

Communications client, peer-to-peer network model, java, broadcast, data distribution

Seznam použitých symbolů a zkratek

ICQ – komunikační klient využívající stejnojmenný protokol

IP – jednoznačný identifikátor síťového rozhraní v síti

J2EE – označení pro enterprise edici javy

J2SE – označení pro standardní edici javy

JDK – Java Development Kit, knihovna pro vývoj v jazyce java

MD5 – Message-Digest algorithm, algoritmus hašovací funkce

TCP – spojově orientovaný protokol transportní vrstvy

UDP – nespojově orientovaný protokol transportní vrstvy

XML – Extensible Markup Language, značkovací jazyk

Obsah

1 Úvod

Komunikace je v současném světě plném informací klíčovým faktorem pro úspěch. Snad každý moderní člověk se již setkal se zařízením, které mu komunikaci usnadnilo. Ať již se jedná o dopis, mobilní telefon či fórum na internetu.

Jen díky předávání informací mohla vzniknout dnes největší informační síť, internet. Pomocí internetu lze v podstatě během vteřiny najít ty nejpodstatnější informace, poslat email, účastnit se či sledovat on-line diskuze a mnoho dalších věcí.

Dnešní informační technologie poskytují řadu možností, jak informace předat. Již standardem je elektronická pošta. Dále existuje mnoho druhů komunikačních programů, více či méně kvalitních, založených na různých technologiích, placené či zdarma.

Většina z nich je však založena na modelu client-server. Tato technologie má mnoho výhod. Již samotné rozdělení úloh v systému odděluje funkční logiku serveru od klienta, který zastává většinou jen úlohu prostředníka. Zajištění identity, bezpečné uložení dat, možnost posílání zpráv doručených až poté, co se příjemce přihlásí, jsou obrovskými výhodami, kterými tento model disponuje. Nevýhoda je jasná. Nutnost mít centrální prvek, na kterém samotný běh závisí.

Komunikačních programů, které pracují na modelu client-server je velké množství. Pro příklad Jabber, ICQ, Windows Live Messenger.

Model, ve kterém nic jako centrální prvek neexistuje, je nazýván peer-to-peer model. Zde se již informace neposílají na server, který by je zpracoval, ale jsou posílány přímo příjemci. Přestože se tento model zdá jednodušší, má mnohá úskalí, která je nutno řešit. Klientů pro takovouto komunikaci mnoho neexistuje. A i těch pár co je se potýká s řadou nevýhod.

Cílem této bakalářské práce je navrhnout a implementovat komunikačního klienta, který by se snažil odfiltrovat nevýhody vyplývající z chybějícího centrálního prvku.

V následujících kapitolách bude rozveden návrh komunikačního klienta v peer-to-peer síti. Dále budou rozebrány vzniklé problémy a nastíněny jejich možná řešení. Nejvýhodnější řešení pak budou popsány podrobněji. V druhé půlce tohoto textu nalezneme postup samotné implementace komunikačního klienta, rozdělení implementace do modulů a jejich popis. Na závěr bude prezentován aktuální stav vývoje.

2 Specifikace požadavků

Nyní budou rozebrány nejdůležitější části zadání. Bude objasněn jejich význam, co pro vývoj komunikačního klienta znamenají a jaké problémy vznikají.

2.1 GUI rozhraní

GUI (Graphics User Interface), neboli grafické uživatelské prostředí je v dnešní době standardní prostředí pro pohodlnou práci běžného uživatele s daným programem. Grafické uživatelské prostředí by mělo být co nejvíce odděleno od funkčního jádra aplikace a proto i při vnitřní změně funkcionality by měla být změna do GUI minimální.

2.2 Nezávislost na serveru

Tato klíčová vlastnost výsledného komunikačního programu je pravděpodobně ta nejtěžší na řešení. Je nutné, aby každý klient mohl navázat komunikaci s jiným, aniž by byly jeho adresa či stav přihlášení získatelné z databáze třetí strany, serveru. Toto je dosti omezující, jelikož komunikační klient bude muset získat tyto informace jiným způsobem.

2.3 Vytváření skupin

Neboli seznam uživatelů, rozdělených do skupin. Z této vlastnosti vyplývá nutnost uchovávání jedinečné identity spolu s dalšími informacemi. Tyto informace je nutné dále distribuovat do všech klientů pro případ, že by se uživatel chtěl přihlásit na jiném počítači. Tato úloha rozhodně není triviální, jelikož při distribuci dat může docházet k různým kritickým situacím.

2.4 Oznámení o příchozí zprávě / oznámení přihlášení uživatelů

Jelikož není možno se dotázat na stav přihlášení uživatele serveru, je potřeba zjistit jeho stav jiným způsobem. Notifikace by měla být přiměřeného charakteru, aby stylem zapadala do GUI aplikace.

2.5 Historie zpráv

Komunikace mezi uživateli by se měla ukládat, aby se na ni později mohli znovu podívat. Historie zpráv by měla být přístupná pouze uživatelům, kteří se komunikace zúčastnili. Jelikož tato vlastnost není kritická, může být tato historie přístupná pouze z počítače, ze kterého daný uživatel komunikoval.

2.6 Šifrování posílaných zpráv

Jelikož přes komunikačního klienta mohou být posílána citlivá data, musí být komunikace mezi uživateli šifrována.

3 Použité metodiky a technologie

3.1 Jazyk pro implementaci

Jelikož pro implementaci desktopových aplikací preferuji staticky typované, objektově orientované programovací jazyky, výběr se zúžil na tři hlavní kandidáty.

C++ - Nejstarší z trojice programovacích jazyků. Překlad probíhá do strojového kódu počítače, obsahuje mnoho knihoven. Existují i multiplatformní knihovny pro vývoj. Neobsahuje garbage collector a složitost vývoje je vyšší oproti zbylým dvěma.

C# - Pro běh vyžaduje .NET Framework. Limitace pouze na Microsoft Windows, i když existuje port .NET Frameworku pro Linux s názvem Mono. Snadná tvorba GUI, obsahuje garbage collector.

Java – Pro běh vyžaduje JRE (Java Runtime Environment). Multiplatformní, obsahuje velmi kvalitní dokumentaci a garbage collector.

Pro implementaci byl nakonec vybrán programovací jazyk Java SE. Pro vývoj byl použit Sun JDK 6 Update 18 a jako vývojové prostředí byly použity NetBeans IDE 6.7.1.

3.2 UML

Pro názornost vysvětlování spojeného s vývojem software je nutné využít metody, které jsou široce uznávány a známy mezi vývojáři. UML, neboli Unified Modeling Language je dnes již standardním prostředkem pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. Díky jejímu rozšíření bude pro názornost využíváno převážně tohoto grafického jazyka.

Pro snadnější orientaci v systému mohou být některé vazby mezi objekty vypuštěny.

4 Analýza vzniklých problémů

4.1 Uchovávání dat

Uchovávání dat je nedílnou a jednou ze základních součástí většiny aplikací. Díky tomuto požadavku existují dnes velmi sofistikované databáze, které se snaží co nejefektivněji otázku uchovávání a zpracovávání dat řešit.

V požadavcích komunikačního klienta je zadáno uchovávání dat uživatele. Konkrétní data, která by si měl komunikační klient ukládat jsou:

- jednoznačná identifikace uživatele
- seznam přátel, rozdělených do skupin
- historie zpráv
- některé další identifikační údaje pro vyhledávání v seznamu uživatelů

Výběr způsobu ukládání dat dosti záleží na druhu, objemu a způsobu jejich využívání.

V případě minimální velikosti, například pro ukládání konfigurace programu, se jako vhodné jeví využít strukturu souboru typu .ini či .properties. Výhoda tohoto typu ukládání dat spočívá v dostupnosti jejich parseru a možnosti číst a editovat tyto data v libovolném textovém editoru. Pro větší množství dat je textový zápis do konfiguračních souborů nevhodný, jelikož tento formát není příliš úsporný. Tento druh ukládání dat bude použit pouze pro uchovávání konfigurace programu.

Pokud bychom hledali elegantnější způsob pro uchovávání dat, jistě by zde stála za zmínku serializace objektů. Tato metoda spočívá v převedení objektu v programovacím jazyce do posloupnosti bytů, jež se uloží například do souboru. Pokud bychom chtěli později objekt ze souboru vyvolat zpět do paměti, provede se takzvaná deserializace a objekt je v paměti vytvořen. Tato metoda má jistě své uplatnění, avšak nezohledňuje nutnost v datech vyhledávat.

Za účelem efektivní úschovy dat a vyhledávání v nich slouží databáze. Existuje jich mnoho druhů. Například objektové databáze, XML databáze a v dnešní době nejrozšířenější, relační databáze. Díky rozšířenosti relačních databází a standardizovanému dotazovacímu jazyku SQL využijeme právě tento druh.

Jelikož je nevhodné, aby spolu se samotným komunikačním klientem bylo nutné instalovat i databázový server, bude ve výsledném komunikačním klientovi použita SQLite databáze. Přestože výkonnost této databáze není excelentní, pro uchování dat zcela postačuje. Pro využití této knihovny bude nutné pouze připojit k projektu soubor s driverem.

4.2 Distribuce dat

V systémech založených na modelu client-server je většina dat uložena na serveru v centrální databázi. V případě nutnosti se pro získání potřebných dat vyšle dotaz na server.

Jelikož však model peer-to-peer žádný centrální databázový server nedovoluje, je nutné využít jiných konstrukcí.

Jedna z možností je data fragmentovat a tyto fragmenty ukládat na jednotlivé uzly. Pokud je potřeba získat data, která daný uzel neobsahuje, vyšle se dotaz na uzel, který požadovaná data spravuje. Nevýhod této metody je mnoho. Velké množství přenášených dat po síti, větší přístupové doby k datům, větší náchylnost ke ztrátě dat při výpadku byť i jediného uzlu. Pro nás největší nevýhoda je však nutnost aktivní spolupráce všech uzlů v síti. Tento problém je dosti zásadní. Z toho vyplývá, že by museli být všichni komunikační klienti neustále spuštěni a připojeni k síti. Toto však nelze garantovat.

Další z metod je duplikace databáze na každý uzel v síti. V případě, že je v síti aktivních více uzlů, tyto uzly si mezi sebou databáze synchronizují. Data se tedy průběžně distribuují do všech uzlů sítě v okamžiku, kdy se pokusí připojit. Databáze samozřejmě musí obsahovat časovou značku záznamů, aby se synchronizovala pouze novější data. Z toho vyplývá, že ve chvíli, kdy se uzel připojí, čili jeho aktivita začíná, má aktuální databázi u sebe. Není tedy nutné se dodatečně na data dotazovat. Výjimka nastává pouze tehdy, pokud se v libovolném uzlu data změní. Tuto změnu je nutné co možná nejrychleji distribuovat do všech připojených uzlů sítě. Přesto tato metoda má několik omezení a kritických stavů, kdy se data nerozšíří na všechny uzly. Pro přiblížení si popíšeme jeden z možných kritických stavů.

Mějme v síti skupinu aktivních, tzn. připojených uzlů. Nyní jeden či více uzlů provede změnu v databázi, která se rozšíří mezi zbylé připojené uzly. Později se všechny aktivní uzly odpojí a v síti není aktivní uzel žádný. Následně se do sítě připojí jeden či více uzlů, které však nepatřily do dříve uvedené skupiny. Přestože jsou tyto uzly aktivní, nemají zcela aktuální databázi. Mohlo by se zdát, že tato situace žádný problém nezpůsobuje, avšak v následujícím odstavci, zabývajícím se identitou, tento problém budeme muset brát v úvahu.

4.3 Identita

Jelikož komunikační klient uchovává jedinečná data, pro každého uživatele jiná, musí být každý uživatel jednoznačně identifikovatelný.

V komunikačních systémech založených na principu client-server je jako jednoznačný identifikační údaj používán tzv. login, případně emailová adresa či telefonní číslo. V praxi to pak znamená, že vlastník e-mailové schránky či telefonního čísla bývá pouze jedna osoba a díky této unikátnosti je zajištěna spojitost s tímto osobním údajem. V případě loginu je tento přihlašovací údaj více anonymní, ale nastává problém, jelikož více osob by mohlo chtít používat stejný login. V tomto případě je registrace pod stejným loginem zamítnuta.

V případě, kdy uvažujeme nad modelem peer-to-peer, je situace složitější. Zde je sice také možno požadovat e-mailovou adresu či telefonní číslo, ale pro ověření, zda-li danému uživateli opravdu daná e-mailová adresa patří, je nutný centrální server. Navíc tento komunikační klient může pracovat ve zcela samostatných sítích, kde v každé síti může existovat uživatel se stejným identifikačním údajem, jelikož se jedná o oddělené systémy, které mezi sebou neinterferují.

Pokud bychom chtěli využít jako identifikační údaj login, nastává problém rozlišit, zda-li v systému již takovýto login neexistuje. V případě centralizované databáze s využitím transakcí toto není problém. Systém lehce zjistí, zda-li v databázi již takovýto login existuje a podle výsledku registraci povolí či zamítne. Bohužel, při absenci centralizované databáze nelze zaručit, že dva klienti v jedné síti budou mít totožný stav databáze. Proto může nastat situace, kdy uživatel A již má ve svém komunikačním klientovi založen účet s loginem „X“, avšak uživatel B nemá zcela aktuální stav databáze, který účet s loginem „X“ ještě neobsahuje. Uživatel B si tedy snadno může účet s loginem „X“ založit a nyní nastává problém. V případě, kdy se systém rozhodne databázi synchronizovat, nastává konflikt, jelikož má dva účty se stejným loginem, avšak tyto účty reprezentují dva různé uživatele. Jelikož login je jednoznačný identifikátor, systém vydedukuje, že se jedná o uživatele jednoho a provede jednu ze zcela nesmyslných věcí. Může buď zachovat starší verzi uživatelského účtu, čímž znemožní situaci, kdy uživatel A jen aktualizoval své údaje, nebo zachová verzi novější, čímž účet uživatele A zcela smaže.

Řešit tento problém lze do jisté míry tím, že při vytvoření uživatelského účtu se přidá k jeho identifikačnímu údaji ještě náhodně vygenerovaný token, který v dostatečné míře odliší jednotlivé uživatele. Nastává však problém, jak uživatele rozlišit pro přihlášení do systému. Jedno z nejrozumnějších řešení se zdá projít všechny uživatelské účty s daným loginem a kontrolovat hesla, zda-li jsou stejná. Robustnost tohoto řešení proti případnému útoku je však velmi chabá. Vždyť si stačí jen v databázi změnit svůj náhodně vygenerovaný token za token požadovaného uživatele a po synchronizaci databáze rázem získáte jeho identitu.

Poslední a pravděpodobně nejspolehlivější metodu pro zajištění jednoznačné identifikace uživatele lze popsat následovně. Identita bude složena jako hash z loginu a hesla zadaného uživatelem. Přestože se tato koncepce nemusí jevit na první pohled zvlášť spolehlivá, při jistých opatřeních se zdá být jako nejvhodnější. V případě požadavku na přihlášení uživatele vyplní uživatel svůj login a heslo. Program nyní z takto zadaných údajů vygeneruje hash a prohledá databázi. V případě, kdy identitu v databázi nalezne, zkontroluje ze záznamu, zda-li souhlasí samotný login a heslo. Heslo lze samozřejmě uložit do databáze také jako hash. Nyní případný útočník pro získání identity musí nejdříve prolomit heslo, ze kterého zná maximálně jen jeho hash. Toto sice není nemožné, avšak velice náročné na čas a znalosti útočníka.

Je zřejmé, že pokud je identita tvořena loginem a heslem, je nutné, aby tyto údaje měly jistou minimální délku. Uživatelů, kteří si vytvoří například účet s loginem „x“ a heslem „x“ může být více. V případě více písmenného loginu a hesla šance na vytvoření uživatelského účtu se stejným jednoznačným identifikátorem rapidně klesá.

Nevýhoda této metody je zřejmá. Uživatelské heslo je neměnné. Tento fakt je sice vážným argumentem proti využití této metody, avšak v případě volby dostatečně silného hesla ztrácí na váze. Navíc touto metodou řešíme situaci, pokud by se potenciální uživatel pokusil přihlásit po změně hesla na počítač, který by ještě obsahoval starou verzi databáze. Takto by uživatel musel zkoušet i starší heslo, což pro běžného uživatele nemusí být triviální záležitost.

4.4 Komunikace v peer-to-peer síti

V předchozím odstavci byl probrán problém s jednoznačnou identifikací jednotlivých uživatelů. Nyní je třeba vyřešit, jak v peer-to-peer síti navázat komunikaci s uzlem dané identity.

V síti s modelem client-server se každý klient musí nejprve autentizovat serveru, který si uchová jeho síťovou adresu po dobu komunikace. Pokud by jiný klient chtěl navázat komunikaci s klientem původním, má možnost tak učinit přes server nebo se alternativně dotázat na síťovou adresu pro navázání komunikace vlastní. V peer-to-peer síti musíme centrální prvek nahradit jinou technikou či přístupem.

Pokud by byla identita svázána se síťovou adresou uzlu, jednoduše by se do databáze připsala i síťová adresa uzlu. Jinému uzlu by následně pro navázání komunikace stačilo pouze nahlédnout do této databáze a zjistit síťovou adresu hledaného uzlu. Problém však nastává při prvotní synchronizaci databáze, kdy uzel nemá v databázi adresy žádných dalších uzlů. To se také dá řešit, avšak přiřazení identity k síťové adrese uzlu je příliš svazující vlastností této metody.

Další metoda využívá tzv. všesměrové vysílání. Toto vysílání pošle data na všechny uzly v síti. Pokud tedy uzel potřebuje navázat komunikaci s uzlem s jinou identitou, jednoduše vyšle všesměrovým vysíláním dotaz na zjištění síťové adresy požadovaného uzlu. Se známou síťovou adresou je nyní možné navázat komunikační kanál.

Přestože se zdá být všesměrové vysílání velmi elegantní, nemělo by se využívat příliš často. Jelikož takto vyslaná data dorazí ke všem uzlům sítě, značně tuto síť zatěžuje. Pro eliminaci přehlcení sítě všesměrovým vysíláním se využívá omezení tohoto vysílání pouze na část sítě. Tato nevýhoda není kritická, protože využití výsledného komunikačního klienta se předpokládá v domovních či místních sítích.

5 Použitá řešení

V následujících odstavcích bude probrán stručný přehled použitých řešení, metodik a technologií pro implementaci komunikačního klienta.

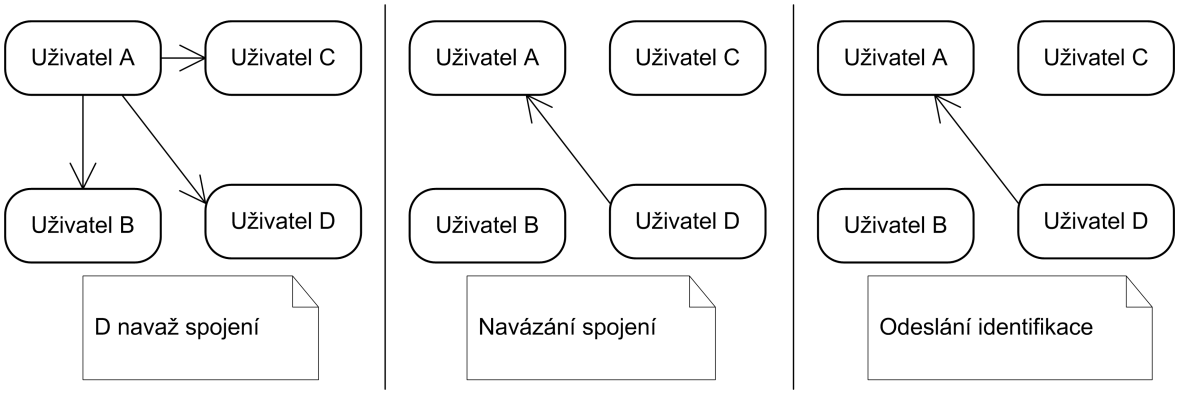
5.1 Komunikace

Pro komunikaci je, dle zadání, nutno použít paketový přenos. Tento způsob komunikace je v dnešní době jeden z nejpoužívanějších. V podstatě většina běžně prodávaných síťových komponent je vyráběna za účelem přenosu dat pomocí paketů.

Jako standardizovaný model pro počítačové sítě se považuje referenční model ISO/OSI [1]. Tento model definuje 7 vrstev, jejichž funkci popisuje z pohledu funkčnosti, nepopisuje však žádnou přímou implementaci.

Ve třetí, síťové vrstvě bude komunikační klient využívat dnes nejznámější protokol IP verze 4. Proč však nevyužít novější specifikaci IP verze 6? Tato technologie ještě není až tak rozšířená a doba, kdy IP verze 6 nahradí IP verzi 4 ještě pár let potrvá. Při správném návrhu a implementaci komunikační části komunikačního klienta nebude při přechodu na IP verze 6 změna zdrojového kódu nějak závratná. Jediný závažný rozdíl IP verze 6 od IP verze 4 je absence broadcastu. Broadcast je v IP verze 6 nahrazen multicastem ke skupině *all-hosts*.

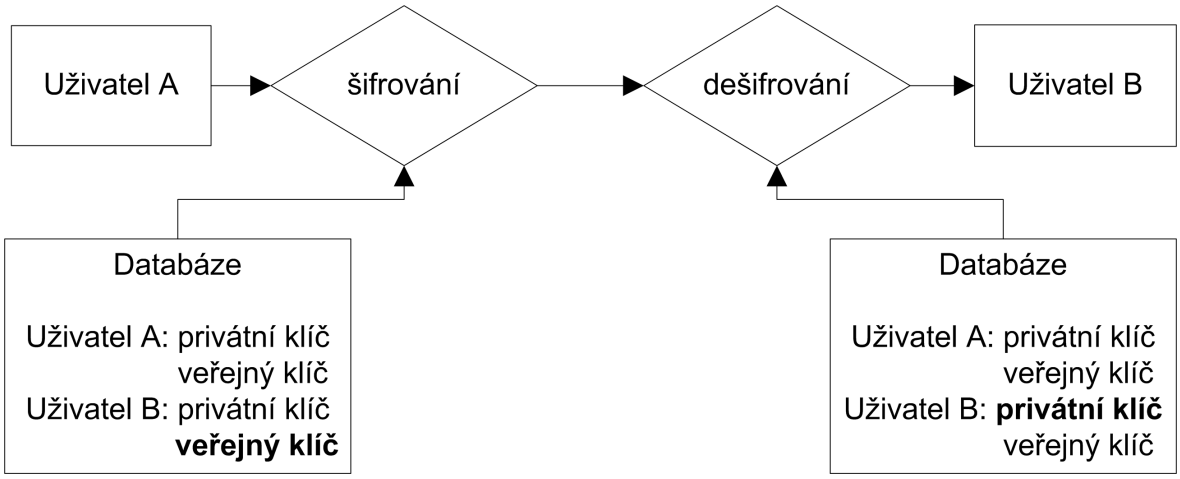
Ve čtvrté, transportní vrstvě budou využívány protokoly dva. První z této dvojice je TCP protokol. Tento protokol je tzv. spojovaný, spolehlivý protokol. Při počátku komunikace se mezi uzly sítě vytvoří virtuální spojení, kterým data procházejí. Spolehlivost tohoto protokolu zajišťuje, že každý odeslaný paket bude doručený příjemci. Pokud se z nějakého důvodu nepodaří spojení navázat, nebo se během přenosu dat přeruší či se paket nepodaří doručit příjemci, nastane chybový stav. V komunikačním klientovi se bude TCP protokol používat pro výslednou komunikaci mezi klienty, kdy už je známa síťová adresa cílového klienta.



Obrázek 1: Navazování spojení

Druhý protokol který bude použit (UDP) zajišťuje nespojovaný, nespolehlivý přenos. Tento protokol nebuduje žádné spojení, ani nekontroluje úspěšnost doručení dat. Hlavní využití tohoto protokolu je v systémech, kterým nezáleží na množství úspěšně přenesených dat, ale na rychlosti doručení. Díky nespojovanému a nespolehlivému charakteru může tento protokol zajistit broadcast a multicast přenosy. A právě broadcast přenos je v komunikačním klientovi využit. Jelikož charakter peer-to-peer sítě nedovoluje zjistit síťovou adresu prvku ze serveru, je použito broadcast vysílání pro její zjištění. Navázání spojení pomocí broadcast vysílání je naznačeno na *„Obrázek 1: Navazování spojení“*.

Nejprve je od stanice A broadcast vysíláním vyslán dotaz všem stanicím, aby stanice s danou identitou D navázala spojení. Pokud stanice D není v síti aktivní, ke spojení nedojde a stanice A po jistém časovém intervalu dojde k rozhodnutí, že stanice D v síti není. Pokud však stanice D v síti existuje a tento dotaz příjme, zjistí si z tohoto dotazu síťovou adresu stanice A a naváže spojení. Toto spojení je již zajištěno protokolem TCP. Posledním krokem je zaslání identity od stanice D stanici A, jelikož se stanicí A může navazovat spojení současně více stanic.



Obrázek 2: Asymetrické šifrování

5.2 Šifrování komunikace

Identita zajišťuje jednoznačnou identifikaci v síti, avšak v popsaném modelu identity není těžké identitu podvrhnout. V podstatě se stačí podívat do databáze a při připojení zaslat identitu, kterou jsme v databázi dříve našli. Nyní by se uživatel mohl vydávat za libovolného uživatele, aniž by o tom kdokoliv věděl. Tomuto případu je nutno zabránit.

Pro šifrování komunikace využijeme asymetrické šifrování. Při asymetrickém šifrování se pro zašifrování a rozšifrování používají jiné klíče. Díky tomu odesílateli stačí znát pouze veřejný klíč pro zašifrování. V běžných systémech si při navazování spojení stanice navzájem vymění

veřejné klíče, pomocí kterých další zprávy šifrují. I kdyby tyto klíče zachytil případný útočník, nemůže pomocí nich rozšifrovat komunikaci probíhající mezi stanicemi.

Do databáze ke každému uživateli jsou přidány dvě nové položky. Jedná se o privátní a veřejný klíč. Privátní klíč je samozřejmě zašifrovaný pomocí hesla uživatele, aby jej nemohl využívat kdokoliv.

Popis principu šifrování v komunikačním klientovi ukazuje *Obrázek 2: Asymetrické šifrování*.

1. Uživatel A dá pokyn komunikačnímu klientovi k zaslání zprávy uživateli B.
2. Komunikační klient vyhledá v databázi veřejný klíč uživatele B a pomocí něj zprávu zašifruje.
3. Proběhne přenos zprávy od komunikačního klienta uživatele A ke komunikačnímu klientovi uživatele B.
4. Komunikační klient uživatele B dešifruje zprávu pomocí svého privátního klíče.
5. Dešifrovaná zpráva je zobrazena uživateli B.

Tento model šifrování řeší jak problém s podvržením identity, tak samotné šifrování komunikace.

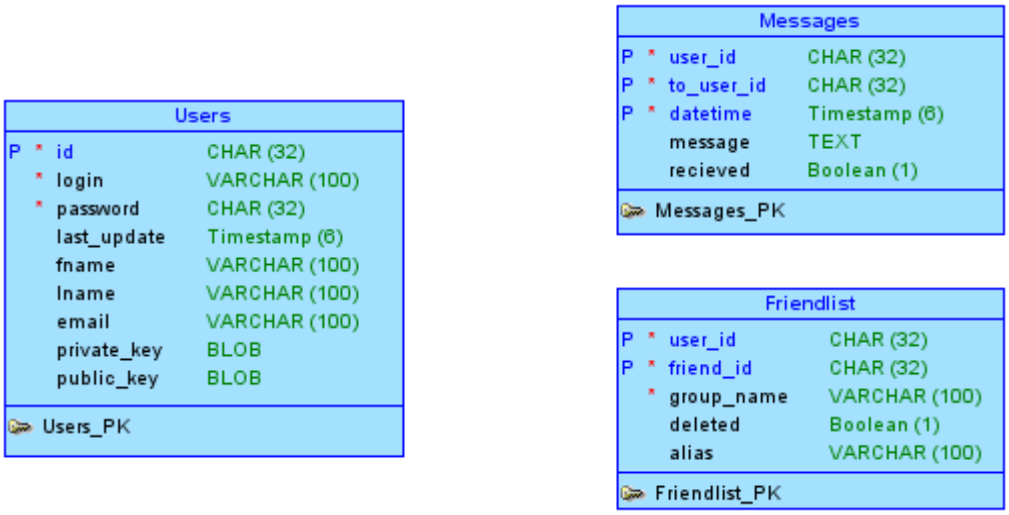
Problém může nastat při absenci záznamu veřejného klíče uživatele B v databázi uživatele A. Možností je buď data nešifrovat a uživateli B zobrazit informaci o nešifrovaném spojení, nebo uživateli A zakázat odeslání zprávy a zobrazení informace o nutnosti synchronizace databáze. Tato situace je však při dobrém návrhu synchronizace téměř vyloučena.

Při synchronizaci je ještě nutné zajistit, aby nebylo možno podvrhnout jiný privátní a veřejný klíč. Řešení je snadné, při synchronizaci se testuje, zda-li jsou tyto hodnoty stejné jako původní. Při rozdílu se jedná od podvrh a synchronizace se přeruší.

5.3 Databáze

Jak již bylo dříve zmíněno, pro uchovávání dat bude použita převážně SQLite databáze. Entity, které bude nutno uchovávat v databázi jsou:

- uživatel
- zprávy odeslané a přijaté od uživatelů
- seznam přátel rozdělených do skupin



Obrázek 3: E-R diagram databáze

5.3.1 Uživatel

Jak již bylo zmíněno v odstavci *4.3 Identita*, každý uživatel by měl mít svou identitu a údaje k ní vázané.

Jako jednoznačný identifikátor id nám poslouží MD5 hash loginu a hesla uživatele. Jelikož by měl být tento identifikátor jedinečný, bude sloužit jako primární klíč. Tento hash v šestnáctkové soustavě zabírá 32 znaků. Dále záznam uživatele v databázi bude muset obsahovat login a heslo uživatele. Maximální velikost loginu, jako většiny doplňkových textových údajů, bude 100 znaků. Ukládání kompletního hesla by bylo nebezpečné, proto budeme ukládat pouze jeho MD5 hash. Dále budeme potřebovat uložit další uživatelské údaje jako jméno, příjmení, či e-mail.

V sekci *4.2 Distribuce dat* byla zmíněna nutnost uchovávat časovou značku jednotlivých záznamů. Přidáme tedy položku pro identifikaci poslední změny údajů.

V sekci *5.2 Šifrování komunikace* byla zjištěna nutnost uchovávat privátní a veřejný klíč. Datovým typem tohoto záznamu budou binární data.

5.3.2 Zprávy

Zprávy odeslané a přijaté by se měly také uchovávat a to jako historie komunikace.

Záznam bude reprezentovat jednu zprávu. Zpráva nemá omezení na délku, proto bude jako datový typ určen řetězec bez omezení délky. Tato zpráva bude samozřejmě šifrována pomocí hesla uživatele. Je nutné uchovávat uživatele, který byl aktuálně přihlášený, respektive který uživatel přijal/odeslal danou zprávu. Dále musíme uchovat i s kým tento uživatel komunikoval. Pro informaci ještě zaznamenáme čas odeslání této zprávy a indikátor, zda-li uživatel zprávu odeslal či přijal.

5.3.3 Seznam přátel

Rozdělení do skupin vyřešíme jako uživatelský list. Budeme uchovávat uživatele, kterého se záznam týká a uživatele, který má být jako záznam v seznamu přátel. Dále uchováme jméno skupiny do které je uživatel přiřazen a přezdívku pro tohoto uživatele.

Vytvoření číselníku se jmény skupin není vhodné, jelikož může nastat stav, že seznam přátel bude distribucí přijat správně, ale tento číselník ne. Z důvodu distribuovaného charakteru dále přidáme ještě indikaci, zda-li je uživatel stále v seznamu přátel.

Výsledné databázové schéma je znázorněno na *Obrázek 3: E-R diagram databáze*.

5.4 Objektově-relační mapování

Jazyk java je objektově orientovaný, databáze SQLite je relační. Tato fakta nám poukazují na nutnost využít dvou různých přístupových technik při programování. Toto však z pohledu znovupoužitelnosti a udržitelnosti kódu není vhodné.

Jako řešení se jeví tyto dvě architektury od sebe co nejvíce oddělit. Jedna z metod je tzv. objektově – relační mapování, která se snaží programátora při implementaci co nejvíce oprostit od relační manipulace s daty.

Způsobů implementace existuje mnoho. Pro příklad může vycházet z návrhového vzoru Active Record. Máme-li tabulku databáze a třídu, jež má řádek této tabulky reprezentovat, musí tato třída implementovat metody pro práci s databází.

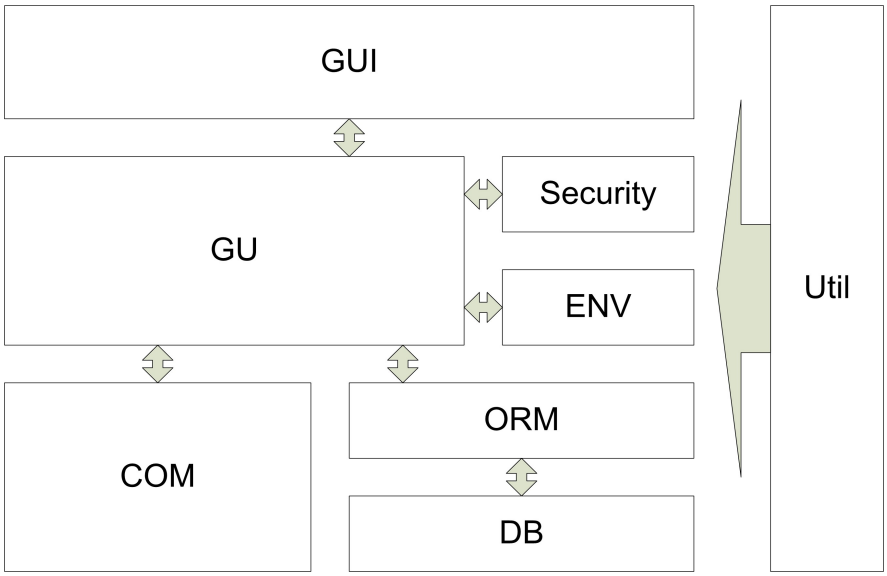
Další návrhový vzor, Data Mapper, má jiný přístup. Třída reprezentující řádek tabulky je tzv. „hloupý objekt“, který data pouze obsahuje a o komunikaci s databází se nestará. Tuto komunikaci zajišťuje objekt nazývaný jako mapper. Na něj poté programátor směřuje požadavky, pokud chce pracovat s daty, uloženými v databázi.

V jazyce java existuje specifikace, která se objektově-relačním mapováním zabývá. JPA, neboli java persistence api, definuje přístupové metody pro práci s databází s pomocí objektů. Bohužel, tato specifikace je obsahem J2EE. Většina nejznámějších implementací této specifikace jsou větší projekty určené na webové servery. Pokud by takováto technologie měla být používána i v komunikačním klientovi, bylo by nutno k výslednému programu přidávat i mnoho knihoven, jež by velikostí jistě mnohonásobně přesáhly kód samotného komunikačního klienta.

Z tohoto důvodu bude implementováno jednoduché objektově-relační mapování, jež zajistí pohodlný objektový přístup. Pro implementaci se bude vycházet ze zjednodušeného návrhového vzoru Data Mapper.

6 Návrh implementace aplikace

V následující části se budeme zabývat návrhem a implementací jednotlivých částí komunikačního klienta. Bude rozebrán jeho návrh a řešení nejkritičtějších částí.



Obrázek 4: Moduly komunikačního klienta

6.1 Moduly systému

Při obecném pohledu na funkce komunikačního klienta je vhodné rozdělení funkcionality na několik autonomních částí, modulů. Rozdělení do modulů zajišťuje lepší udržovatelnost, znovupoužitelnost kódu a rozděluje základní funkcionalitu software do jednotlivých celků.

Na *„Obrázek 4: Moduly komunikačního klienta“* je znázorněno rozdělení do modulů komunikačního klienta.

Moduly pro komunikačního klienta jsou následující:

- COM – komunikační modul
- DB – databázový modul
- ORM – modul pro objektově-relační mapování
- ENV – modul pro uchovávání stavu aplikace
- GU – modul pro zajištění základní funkcionality komunikačního programu
- Security – modul pro práci se šifrováním

- GUI – modul pro grafické zobrazení komunikačního klienta
- Util – modul pro nezařaditelné pomocné třídy

Těchto sedm modulů zajistí dostatečnou pružnost pro vývoj aplikace a jejím případným úpravám.

Na nejnižší vrstvě nalezneme moduly zajišťující komunikaci s okolím.

COM (komunikační modul) odstiňuje zbylé moduly od nutnosti starat se o způsob komunikace s okolními klienty. Nutnost interakce okolních modulů by měla být co nejmenší, aby v případě změny způsobu komunikace s okolními klienty musel být změněn pouze tento modul. Funkcionálně by tento modul měl být schopen odeslat a přijímat zprávy, případně informovat o nastalé chybě.

DB (databázový modul) zajišťuje možnost uložení dat na definované úložiště. Má za úkol zkontrolovat integritu databáze, případně vytvořit novou. Jako modul by měl být schopen odstínit okolní moduly od nutnosti starat se o stav databáze a připojení k ní.

ORM (modul pro objektově-relační mapování) je část aplikace, která odstíní zbytek od nutnosti práce s relační databází. Zajišťuje funkce pro práci s daty ostatních částí aplikace.

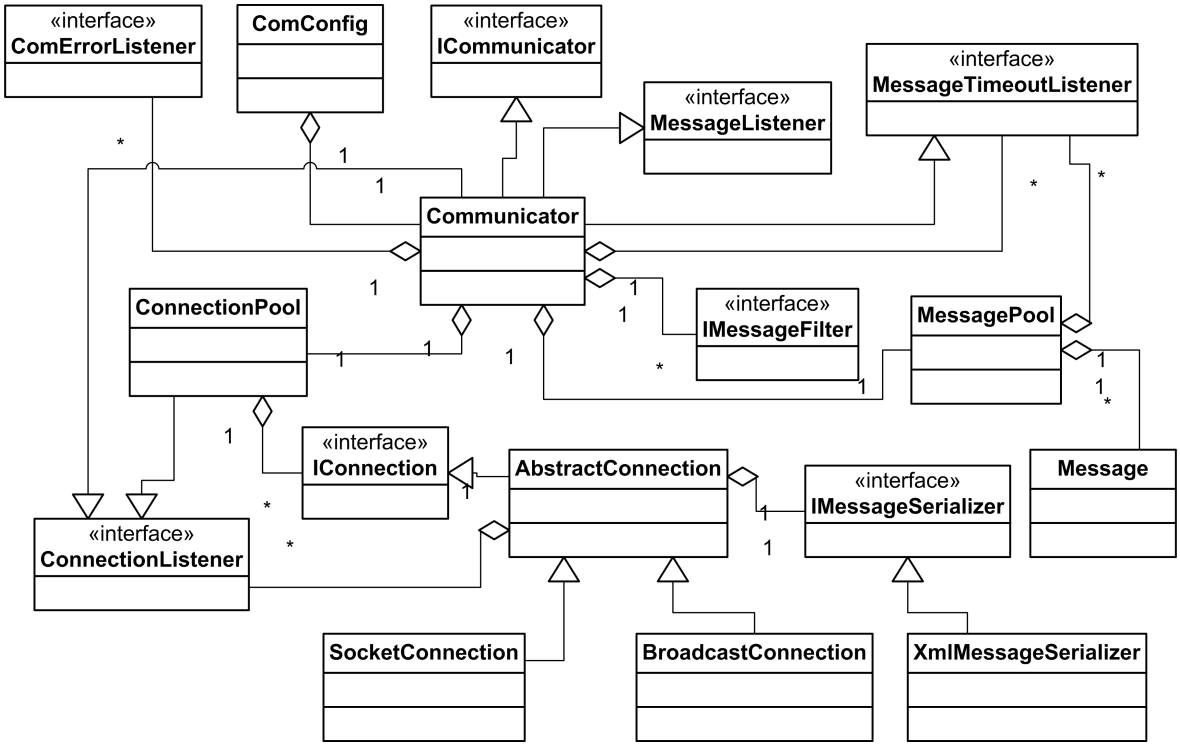
ENV (modul pro uchování stavu aplikace) uchovává data významná pro celou aplikaci. Obsahuje nastavení, aktuální data uživatele, objekty zajišťující určité služby pro aplikaci a podobně.

GU (centrální modul) se stará o spolupráci většiny okolních modulů. Jako jediný by měl komunikovat s grafickým modulem pro co nejlepší odstínění funkcionality programu od grafického rozhraní. Navíc zajišťuje základní funkcionalitu komunikačního programu jako například stav contactlistu, synchronizace databáze, ukládání historie a další.

Security je pomocný modul starající se o šifrování dat a další funkce spojené s bezpečností komunikace.

GUI (modul obsahující definici grafického uživatelského rozhraní) definuje rozhraní mezi uživatelem a funkcionalitou komunikačního klienta.

Util je modul pro části aplikace, které nelze jednoznačně zařadit a jsou využívány ve více modulech.



Obrázek 5: Třídní diagram komunikačního modulu

6.2 Komunikační modul

Jak již bylo napsáno dříve, komunikační modul obsahuje funkcionalitu pro odstínění ostatních modulů od nutnosti obstarávat si komunikaci sami. Definuje jakési abstraktní rozhraní, přes které ostatní moduly komunikují a které tento modul sám implementuje. Pokud by v budoucnu byl požadavek na změnu způsobu komunikace, například požadavek na komunikaci přes IPv6, je možné implementaci změnit bez nutnosti zásahu do ostatních modulů.

Na *„Obrázek 5: Třídní diagram komunikačního modulu“* je znázorněn třídní diagram komunikačního modulu. Jako výše zmíněné rozhraní pro definici metod, které komunikační modul bude nabízet, je zde rozhraní ICommunicator.

Definice tohoto rozhraní je následující:

```
public interface ICommunicator extends Runnable {
    public void requestConnection(String key);
    public void closeConnection(String key);
    public void sendMessage(Message message);
    public void addMessageListener(MessageListener listener);
    public void removeMessageListener(MessageListener listener);
    public void addComErrorListener(ComErrorListener listener);
```

```
    public void removeComErrorListener(ComErrorListener listener);
    public void addMessageFilter(IMessageFilter filter);
    public void removeMessageFilter(IMessageFilter filter);
    public void addMessageTimeoutListener(MessageTimeoutListener
      listener);
    public void removeMessageTimeoutListener(MessageTimeoutListener
      listener);
    public IMessageSerializer getMessageSerializer();
    public void close();
    public boolean isClosed();
}
```

Vidíme zde řadu metod, pomocí kterých okolní moduly komunikují. Metody *requestConnection* a *closeConnection* jsou volány nepovinně a jsou zde pouze pro rychlejší odezvu uživateli. Tyto metody se využívají například při otevření okna pro komunikaci s daným uživatelem. Komunikační klient tím již připraví spojení a zjistí případnou nepřítomnost uživatele v síti. Následující metoda *sendMessage* je nejpoužívanější ze všech zde uvedených. Pomocí této metody se pokusí komunikační modul odeslat zprávu uživateli, který je uveden jako parametr v této zprávě. Následující metody slouží k přidávání a odebírání posluchačů událostí. Tento způsob propagace událostí vychází z návrhového vzoru *Observer*. První uvedená dvojice *addMessageListener* a *removeMessageListener* přidává a odebírá posluchače pro propagaci přijatých zpráv od vnějšího prostředí, od ostatních komunikačních klientů. Další z dvojic metod *addComErrorListener* a *removeComErrorListener* zajišťuje přidávání a odebírání posluchačů událostí obsahující informaci o chybových stavech komunikačního klienta. Poslední dvojice *addMessageTimeoutListener* a *removeMessageTimeoutListener* zajišťuje přidávání a odebírání posluchačů událostí obsahující informace o nedoručení zprávy. Metoda *getMessageSerializer* navrátí serializer zpráv. Poslední metody *close* a *isClosed* ukončují funkci, respektive zjišťují, zda-li funkce již byla ukončena.

Nyní bude popsána funkce samotné implementace tohoto rozhraní třídy *Communicator*. Třída *Communicator* mimo posluchačů událostí obsahuje tři základní závislosti na objektech. Třídy těchto objektů jsou *ComConfig*, *ConnectionPool*, *MessagePool*.

ComConfig obsahuje definici parametrů pro funkci komunikačního modulu. Proměnné, které tato třída obsahuje, jsou následující:

```
private String key = null;
private String serverIp = null;
private String broadcastIp = null;
private int serverPort = -1;
private int broadcastLocalPort = -1;
private int broadcastForeighPort = -1;
```

```
private long messageTimeout = 5000;
private long unusedConnectionTimeout = 30000;
```

Vidíme, že se jedná převážně o nastavení IP adres a portů. Proměnná *key* obsahuje identifikátor uživatele, který daný modul využívá. Následuje IP adresa, na které se má server vytvořit, broadcast adresa sítě, port na kterém má server naslouchat, lokální port pro naslouchání broadcast zpráv a vzdálený port, na který se mají broadcast zprávy odesílat. Následují poslední dvě proměnné. Proměnná *messageTimeout* definuje čas v milisekundách, po kterém se má zpráva stát nedoručenou. Poslední, *unusedConnectionTimeout* definuje čas v milisekundách, po kterém se má uzavřít spojení, pokud není využíváno. Objekt třídy *ComConfg* se předává do konstruktoru třídy *Communicator* a všechny výše uvedené parametry se dají nastavit pomocí přístupových metod.

Třída *ConnectionPool* se stará o správu spojení. Tato třída obsahuje následující proměnnou:

```
private final Map<String, List<ConnectionWrapper>> connections =
    new HashMap<String, List<ConnectionWrapper>>();
```

Tato proměnná je hash mapa, jejíž klíče jsou identifikátory uživatelů, se kterými jsou spojení uložena. Jako hodnota této mapy je list objektů třídy *ConnectionWrapper*. Třída *ConnectionWrapper* obaluje spojení *IConnection* a dále datum a čas poslední uskutečněné komunikace. Pomocí této časové známky následně může třída odstraňovat spojení, které se nepoužívá. O toto odstranění se stará vnitřní třída *ConnectionPoolTimerTask*, jejíž metodu *run* volá předem definovaný časovač.

Hodnota mapy není pouze jediný *ConnectionWrapper*, ale množina těchto objektů. Jelikož není zaručeno, že současně nebude navázáno více spojení, je nutné uchovávat všechny. Pokud by nebyly uchovávány všechny spojení, v kritických situacích by mohlo dojít ke ztrátě dat.

Třída *MessagePool*, jak již název napovídá, se stará o uchování dosud neodeslaných zpráv. Struktura této třídy je velice podobná třídě *ConnectionPool*. Seznam zpráv je definován v následující proměnné:

```
private Map<String, List<MessageBox>> messagePool =
    new HashMap<String, List<MessageBox>>();
```

Zprávy jsou uloženy v mapě, jejíž klíč je opět identifikátor uživatele, kterému se má zpráva odeslat a hodnota je seznam objektů třídy *MessageBox*. Třída *MessageBox* obaluje třídu zprávy, *Message* a navíc obsahuje časové razítko vložení této zprávy do seznamu. Následně tato třída kontroluje, zda-li zpráva není již ve frontě příliš dlouho a v případě že ano, tato třída vyvolá událost *messageTimeout*. Tato událost je vyvolána u zaregistrovaných posluchačů implementujících rozhraní *MessageTimeoutListener*. V komunikačním modulu je jako posluchač zaregistrována třída *Communicator*, jež tuto událost dále propaguje a informuje posluchače o nemožnosti doručit tuto zprávu.

Další z objektů, jež třída *Communicator* obsahuje, jsou objekty implementující rozhraní *IMessageFilter*. Rozhraní *IMessageFilter* definuje následující dvě metody:

```
public Message filterBeforeSend(Message message);
public Message filterBeforeReceive(Message message);
```

První z uvedených metod je volána před odesláním zprávy. Druhá je volána po přijetí zprávy, avšak ještě před distribucí do ostatních modulů. Filtr může zprávu libovolně upravovat a výsledek navrátí jako návratovou hodnotu. V případě, že by zpráva měla být zahozena, může filtr vrátit hodnotu *null* a od dalšího zpracování této zprávy je upuštěno. V komunikačním klientovi se tyto filtry využívají například k šifrování zpráv či ukládání zpráv do historie.

Po vytvoření třídy *Communicator* a jeho spuštění jako nové vlákno se v tomto objektu vytvoří selektor síťových spojení, zajišťující neblokující přístup ke spojením, což umožňuje přijímat a odesílat data ve více spojeních v jediném vlákně. Po vytvoření selektoru se vytvoří serverový socket na ip adrese a portu definovaném v objektu třídy *ComConfig*. Tento serverový socket se poté přidá do selektoru.

Následně se vytvoří broadcast spojení, u něhož se zaregistruje *Communicator* jako posluchač zpráv (*MessageListener*) a stavu událostí (Connection*Listener*). Toto spojení se také zaregistruje do selektoru a přidá se do objektu třídy *ConnectionPool*.

Následuje smyčka, ve které se kontroluje požadavek jednotlivých spojení na interakci. V případě kladné odpovědi se provede jeden z následujících úkonů: příjem zprávy, odeslání zprávy, spojení přijato či spojení navázáno.

Rozhraní *IConnection* definuje metody pro práci se třídou jako se síťovým spojením, jež komunikační modul využívá. Komunikační modul implementuje dva základní druhy spojení, TCP spojení a broadcast spojení. Implementací TCP spojení lze nalézt ve třídě *SocketConnection*, broadcast spojení ve třídě *BroadcastConnection*. Základní funkce těchto tříd je sjednotit a ulehčit odesílání a příjem zpráv objektu třídy *Message*. Obě tyto třídy mají společného předka *AbstractConnection*, jež definuje společné metody obou těchto spojení. Každý objekt těchto tříd obsahuje kanál, jakožto java implementaci síťového spojení.

Třída *SocketConnection* obsahuje buffer přijatých znaků, jež poté převádí na zprávu.

Třída *BroadcastConnection* však obsahuje buffer pro každou IP zvlášť, jelikož při fragmentaci zprávy a případném průniku více zpráv od více odesílatelů by došlo k nekonzistenci XML zprávy a data by byla zahozena. Tato situaci díky více bufferům však nastat nemůže.

Každá instance obsahuje objekt implementující rozhraní *IMessageSerializer*, jež dokáže převést objekt třídy *Message* na řetězec znaků.

Jediná třída implementující rozhraní *IMessageSerializer* v komunikačním modulu je třída *XmlMessageSerializer*. Tato třída převádí objekt třídy *Message* na řetězec a zpět, jež je ve formě XML. Pro efektivní implementaci této třídy byla využita technologie JAXB, jež dokáže převádět

mezi objektem a jeho XML reprezentací. Při převodu řetězce znaků do objektu je nutné odfiltrovat část řetězce, která nedefinuje objekt třídy *Message*. K tomuto účelu jsou zadefinovány počáteční a koncové značky XML řetězce.

Pro přenos zprávy mezi komunikačním modulem a ostatními moduly je v komunikačním modulu připravena třída *Message*. Instance této třídy má představovat informaci, jež má být předána dále. Proměnné, které tato třída obsahuje, jsou následující:

```
protected String type;
protected String message;
protected Object object;
protected Object[] objects;
protected String receiver;
protected int version = Environment.getInstance().getVersion();
protected Date date;
protected String sender;
protected String senderIp;
protected int senderPort;
protected boolean crypted = false;
```

Pro větší přehlednost byly z definice proměnných odstraněny anotace *@XmlElement*, jež jsou nutné pro technologii JAXB, pro serializaci do XML dokumentu.

První polovina těchto proměnných slouží pro předávání informací. Tyto proměnné jsou nastavovány ostatními moduly a s těmito proměnnými dále pracují. Druhá polovina, začínající proměnnou *version*, obsahuje informace pro správné doručení zprávy. Tyto proměnné využívá výhradně komunikační modul.

Proměnná *type* obsahuje řetězec označující typ zprávy. Moduly při posílání zpráv mohou poté do této proměnné zapsat identifikátor umožňující ostatním modulům rozlišit, zda-li tato zpráva je směřována jim, či ne. Seznam použitých typů zpráv je následující:

```
public static final String TYPE_UNKNOWN = "UNKNOWN";
public static final String TYPE_MESSAGE = "MESSAGE";
public static final String TYPE_MY_IDENTITY = "MY IDENTITY";
public static final String TYPE_CONNECT_ME = "CONNECT ME";
public static final String TYPE_MY_DATABASE_INFO =
    "MY DATABASE INFO";
public static final String TYPE_MY_DATABASE_HASH =
    "MY DATABASE HASH";
public static final String TYPE_MY_DATABASE_USER =
    "MY DATABASE USER";
public static final String TYPE_SEND_DATABASE_USER =
```

```
    "SEND DATABASE USER";
public static final String TYPE_I_AM_ONLINE_NOTIFICATION =
    "I AM ONLINE";
public static final String TYPE_LOGGING_OFF = "LOGGING OFF";
```

Typ *TYPE_UNKNOWN* reprezentuje zprávu, jejíž typ nebyl definován. Zbylé typy budou průběžně popsány v kapitolách o modulech, které tyto typy zpráv využívají.

Další z proměnných třídy *Message*, je proměnná s názvem *message*. Tato proměnná, jak již název napovídá, obsahuje zprávu, neboli významnou informaci pro daný typ zprávy.

Následující proměnné *object* a *objects* jsou zde pro případ, že by bylo nutno posílat i objekty jiného typu. Pokud jsou tyto proměnné využívány pro posílání objektů, musí být třídy těchto objektů správně anotovány. Anotace jsou nutné pro správnou serializaci do XML dokumentu. Více o anotacích JAXB technologie naleznete na [2].

Poslední proměnná, kterou využívají vnější moduly, je proměnná *receiver*. Do této proměnné je nutné vložit identifikátor příjemce zprávy. Pokud by měla být zpráva odeslána všem komunikačním klientům, je nutné do této proměnné vložit konstantu *Message.RECEIVER_ALL*.

Proměnná *version* definuje verzi zprávy, *date* obsahuje datum vytvoření zprávy, *sender*, *senderIp* a *senderPort* obsahují informace o odesílateli. Poslední proměnná *crypted* obsahuje informaci o tom, zda-li je obsah proměnné *message* šifrován.

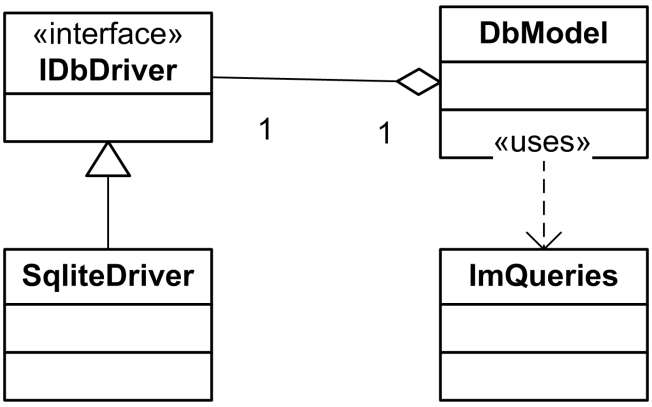
V případě, že komunikační modul dostane podnět pro navázání komunikace, pokusí se navázat spojení se zadaným klientem. Princip, jak se navazuje spojení s využitím peer-to-peer modelu, byl již vysvětlen v *5.1 Komunikace*. Nyní bude popsáno, jak tento princip komunikační modul implementuje.

1. Komunikační modul klienta A dostane podnět, pro navázání komunikace s klientem B.
2. Komunikační modul klienta A vytvoří zprávu typu *TYPE_CONNECT_ME*, jako příjemce nastaví konstantu *Message.RECEIVER_ALL* a odešle tuto zprávu broadcast spojením všem naslouchajícím klientům.
3. Komunikační modul klienta B tuto zprávu přijme. Z proměnné *sender* zjistí příjemce a vytvoří třídu *SocketConnection*, reprezentující spojení. Tuto třídu zaregistruje v selektoru a vloží ji do úložiště spojení třídy *ConnectionPool*. Následně vytvoří zprávu typu *TYPE_MY_IDENTITY* a odešle ji nově vytvořeným spojením.
4. Komunikační modul klienta A příjme spojení, vytvoří také instanci třídy *SokcetConnection* a vloží jej do selektoru.
5. Komunikační modul klienta A přijme zprávu odeslanou klientem B v kroku 3 a vloží spojení do třídy *ConnectionPool*. Vloží jej pod identitou, jež je v proměnné *sender* přijaté zprávy.

Nyní bude popsáno, jak se komunikační klient zachová, pokud dostane podnět na odeslání zprávy.

Na instanci třídy *Communicator* je zavolána metoda *sendMessage*, které je jako parametr předán objekt zprávy, jež se má přenést. *Communicator* do této zprávy vloží údaje o odesílateli. Ze zprávy z proměnné *receiver* si zjistí identifikační údaj příjemce. Následně ověří, zda-li instance třídy *ConnectionPool* již obsahuje spojení s daným uživatelem. Pokud ne, nastane proces navazování komunikace, jež je popsán výše a zpráva je vložena do instance třídy *MessagePool*. V okamžiku, kdy s daným uživatelem spojení existuje, jsou na zprávu aplikovány filtry a je daným spojením odeslána.

V případě, že do určitého časového intervalu není zpráva odeslána, je delegována informace všem zaregistrovaným posluchačům třídy *MessageTimeoutListener*.



Obrázek 6: Třídní diagram databázového modulu

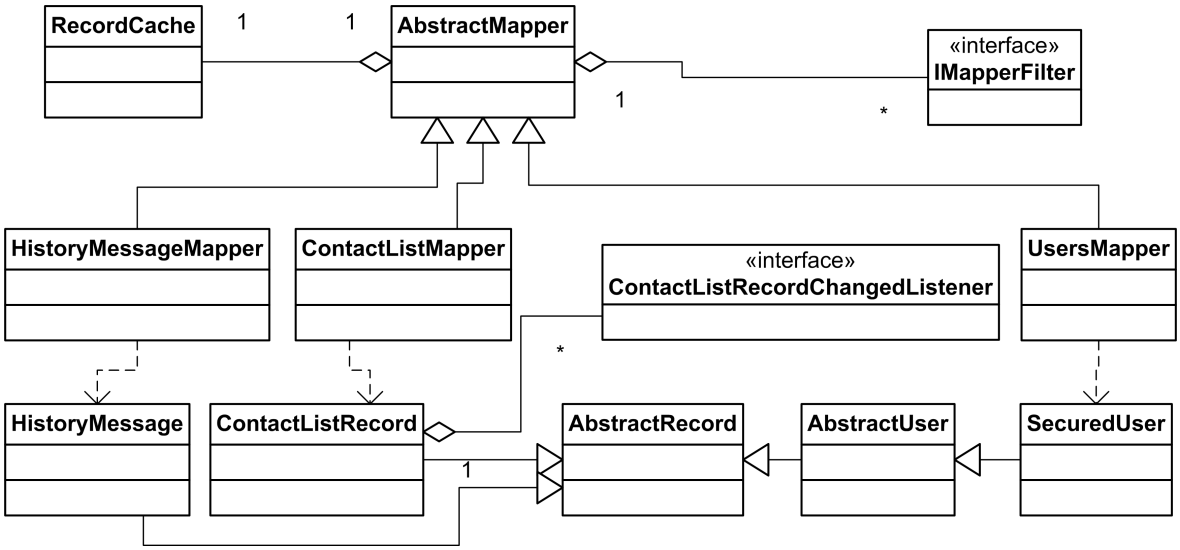
6.3 Databázový modul

Struktura databázového modulu je velmi strohá. Znázorněna je na *„Obrázek 6: Třídní diagram databázového modulu“*. Pro zobecnění přístupu do databáze je vytvořeno rozhraní *IDbDriver*. Toto rozhraní v sobě zahrnuje metody pro navázání spojení, vytvoření databáze a podobné.

Implementaci nalezneme ve třídě *SqliteDriver*, jež tyto metody implementuje pro databázi SQLite. O databázi SQLite se lze dočíst na [3]. Pro správnou funkci třídy *SqliteDriver* je nutné připojit knihovnu *sqlitejdbs*. O této knihovně lze nalézt více na [4].

Třída *DbModel* v sobě zahrnuje funkcionalitu pro kontrolu integrity a vytvoření nové databáze. Zajišťuje, že databáze je v konzistentním stavu a lze s ní pracovat.

Pomocná třída *ImQueries* obsahuje dodatečné informace nutné pro vytvoření databáze.



Obrázek 7: Třídní diagram modulu objektově-relačního mapování

6.4 Modul objektově-relačního mapování

Modul objektově-relačního mapování obstarává odstínění okolních modulů od nutnosti pracovat s databází. Třídní diagram tohoto modulu lze nalézt na *„Obrázek 7: Třídní diagram modulu objektově-relačního mapování“*.

Tento modul implementuje objektově-relační mapování pro tři tabulky reprezentující uživatele (*Users*), záznam ze seznamu přátel (*Friendlist*) a zprávu historie (*Messages*). Pro každou z těchto tabulek je třída zde reprezentující jeden záznam tabulky a třída reprezentující prostředníka, který mapování provádí.

Pro třídu mapující objekty je zde abstraktní předek *AbstractMapper*. Tato třída obsahuje metody společné pro všechny mapující třídy a deklaruje metody, které musí být implementovány. Obsahuje také seznam objektů tříd, implementující rozhraní *IMapperFilter*. Tyto objekty (filtry) obsahují metody pro úpravu dat před uložením a po načtení z databáze. Tohoto je dále využíváno pro šifrování zpráv historie.

Jelikož výsledný komunikační klient je interaktivní aplikace, je nutné, aby provedené změny byly okamžitě aktualizovány na všech místech. Pokud by byly z databáze načteny dva objekty, oba reprezentující stejný záznam a jeden z nich by byl změněn, druhý by obsahoval stále starou hodnotu. V tomto případě by objekt obsahující starou hodnotu obsahoval již neplatná data, které by se mohla uživateli zobrazovat.

Možné řešení je například propojit tyto záznamy podle návrhového vzoru Observer, ale toto není moc praktické. Použité řešení je propagovat do ostatních modulů vždy jen jednu instanci pro každý záznam. Seznam používaných záznamů je ukládán v instanci třídy *RecordCache*.

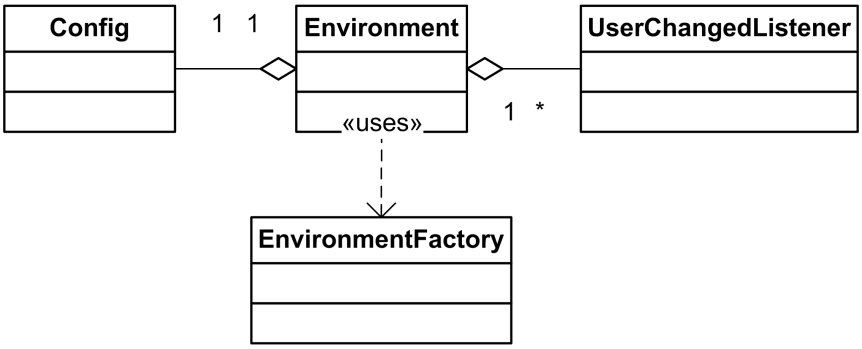
Třída *RecordCache* obsahuje následující proměnnou:

```
protected Map<T, WeakReference<T>> cache =
    new WeakHashMap<T, WeakReference<T>>();
```

Takto definovaná proměnná (mapa) obsahuje aktuálně užívané objekty. *T* je generický datový typ. Jako implementaci mapy je zde použita *WeakHashMap*. Jedná se o mapu, jejíž klíč je ukládán slabou referencí. To znamená, že pokud klíč (objekt) již není nikde jinde používán, je z mapy smazán a následně úplně odstraněn i z paměti. Jako hodnota této mapy je instance třídy *WeakReference* obsahující daný záznam. Hodnota mapy nemůže být záznam samotný, jelikož by na tento záznam existovala silná reference a objekt by díky tomu nemohl být odstraněn.

Potomci třídy *AbstractMapper*, což jsou *HistoryMessageMapper*, *ContactListMapper* a *UsersMapper* obsahují metody pro práci se záznamy. Všechny tyto třídy implementují metody *find*, *insert* a *update*. Toto jsou tři základní přístupové metody pro práci s tabulkami. Dále každý poté implementuje další metody pro vyhledávání záznamů.

Jelikož některá data uživatele jsou šifrována a aktuálně přihlášený uživatel má tyto hodnoty uloženy v nešifrované podobě, je nutno takovou odlišnost brát v potaz. V tomto modulu je proto definována abstraktní třída *AbstractUser*, která zahrnuje proměnné, jež se nešifrují. Dále je zde třída *SecuredUser*, která reprezentuje uživatele se šifrovanými daty. Třídu pro uživatele s dešifrovanými údaji nalezneme v modulu pro zajištění základní funkcionality komunikačního programu. Třída se nazývá *User* a má konstruktor s typy parametrů *SecuredUser* a *String*. První definuje záznam se zašifrovanými údaji, druhý pak heslo, kterým se mají tyto údaje dešifrovat.



Obrázek 8: Třídní diagram modulu pro uchování stavu aplikace

6.5 Modul pro uchování stavu aplikace

Modul pro uchování stavu aplikace je jakýsi centrální bod, ve kterém jsou uchovávány aktuální hodnoty nastavení komunikačního klienta při běhu. Třídní diagram tohoto modulu lze nalézt na *„Obrázek 8: Třídní diagram modulu pro uchování stavu aplikace“*.

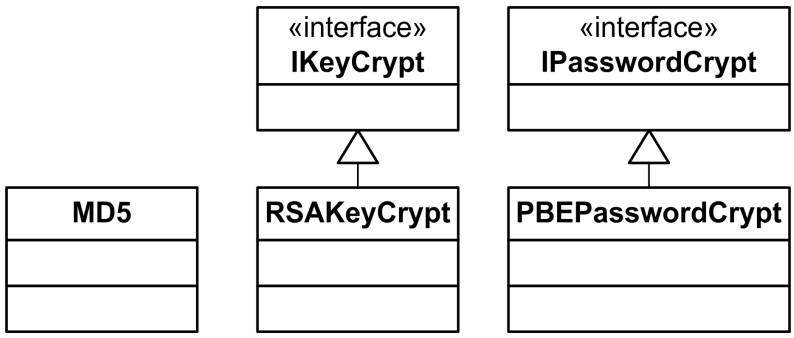
Tento modul je vcelku jednoduchý. Pro nastavení aplikace slouží instance třídy *Config*. Do tohoto objektu jsou nejprve načteny hodnoty konfigurace a poté je předán do objektu třídy *Environment*. Třída *Environment* má na starosti uchování používaných prostředků. V této třídě nalezneme zprostředkovatele pro práci s databází, aktuálního uživatele či objekty zprostředkovávající šifrování.

Třída *Environment* je koncipována jako singelton. Toto je zajištěno privátním konstruktorem a ve statickém inicializátoru se vytvoří instance této třídy, která je pak veřejně dostupná přes metodu

```
public static Environment getInstance()
```

Pro inicializaci objektů v singeltonu *Environment* je použita továrna *EnvironmentFactory*. V této továrně se inicializují objekty, které pak dále poskytují služby dalším modulům.

V případě, kdy okolní moduly potřebují zareagovat na změnu uživatele, je zde možnost zaregistrovat posluchače, který implementuje rozhraní *UserChangedListener*.



Obrázek 9: Třídní diagram modulu pro práci se šifrováním

6.6 Modul pro práci se šifrováním

Modul pro práci se šifrováním definuje rozhraní a prostředky, díky nimž lze jednoduše šifrovat text.

Rozhraní *IKeyCrypt* definuje metody, jež musí implementovat třída, která má zajišťovat asymetrické šifrování pomocí klíče. Definované metody jsou následující:

```
public String encrypt(String source, PublicKey key)
    throws  InvalidKeyException;
public String decrypt(String source, PrivateKey key)
    throws InvalidKeyException;
public KeyPair generateKey()
    throws NoSuchAlgorithmException;
```

Metody *encrypt* a *decrypt* zde slouží pro šifrování, respektive dešifrování zprávy pomocí zadaného klíče. V případě chyby při zpracování lze vyvolat výjimku *InvalidKeyException*.

Následuje metoda *generateKey*, pomocí níž lze vygenerovat novou dvojici klíčů, obalenou do třídy *KeyPair*.

Třídu implementující toto rozhraní, kterou komunikační klient využívá, je třída *RSAKeyCrypt*. Tato třída, jak již z jejího názvu vyplývá, využívá algoritmus RSA pro asymetrické šifrování. Více o tomto algoritmu lze nalézt na [5].

Rozhraní *IPasswordCrypt* definuje metody, jež musí implementovat třída, která má zajišťovat šifrování pomocí hesla. Definované metody jsou následující:

```
public String encrypt(String source, String key)
    throws InvalidKeyException;
public String decrypt(String source, String key)
    throws InvalidKeyException;
public String getKeyHash(String key);
public boolean isValidKey(String key, String keyHash);
```

Metody *encrypt* a *decrypt*, jako v předchozím případě, šifrují, respektive dešifrují zadaný řetězec. Nyní je však pro šifrování užito heslo.

Metodou *getKeyHash* lze vygenerovat hash hesla, který je předán jako parametr. Pomocí poslední metody (*isValidKey)* lze zjistit, zda-li je daný hash vygenerovaný z daného klíče.

Jako implementace je zde třída *PBEPasswordCrypt*. Tato třída provádí šifrování pomocí metody PBEWithMD5AndDES a pro generování hashe používá metodu MD5.

6.7 Modul pro nezařaditelné pomocné třídy

Tento modul obsahuje pomocné třídy, využíváné v ostatních modulech, ale nereprezentují funkcionalitu daného modulu.

Za zmínku zde stojí uvést dvě třídy. *ComparatorHelper* a *PropertiesMapper.*

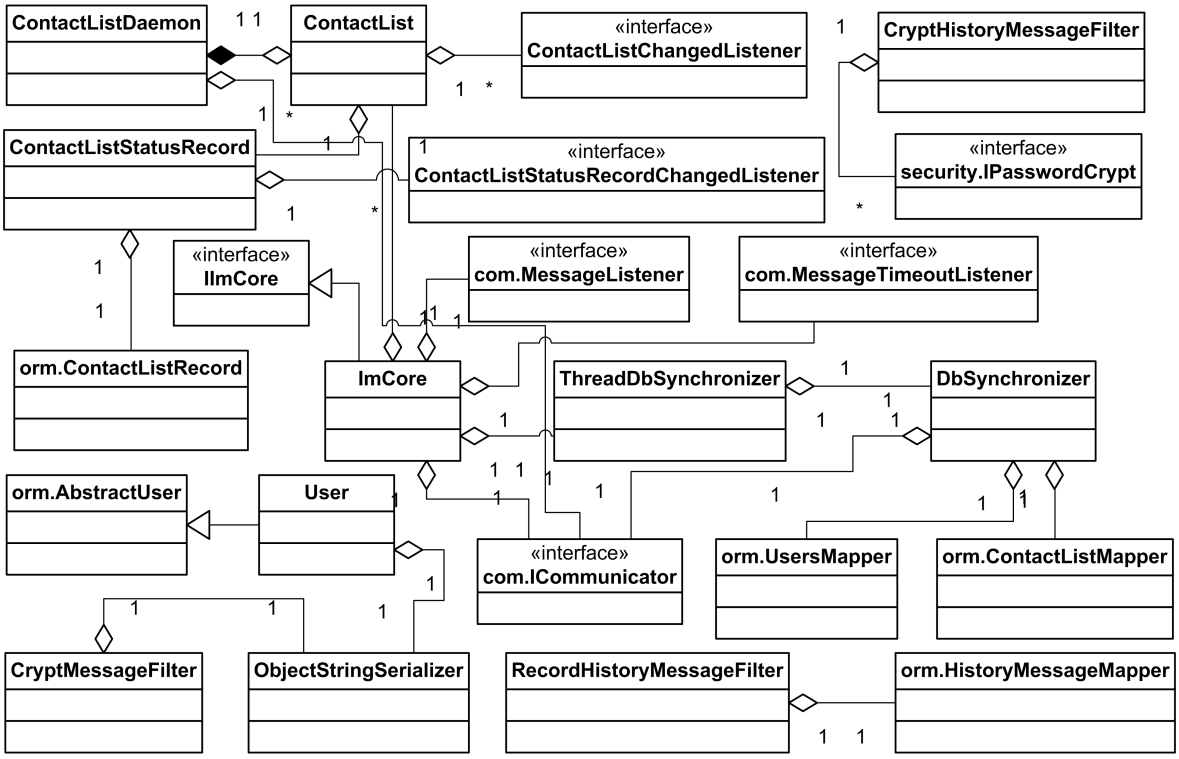
První, *ComparatorHelper*, usnadňuje implementaci rozhraní *Comparable*. Obsahuje statickou metodu s následující definicí:

```
public static int compare(Comparable[] objects1,
    Comparable[] objects2)
```

Tato metoda projde postupně všechny objekty předané v parametrech *objects1* a *objects2* a provádí na nich metody *compareTo*. V případě, že narazí u některé dvojice k nenulovému výsledku, navrátí tuto hodnotu.

Druhá třída, *PropertiesMapper*, provádí mapování mezi properties souborem a objektem. Struktura properties souboru je velice podobná struktuře ini souboru.

Pro deklaraci, které parametry se mají mapovat, se využívá anotace *MappedProperty*. V případě předání properties souboru a zavolání metody *setMappedProperties* se automaticky do objektu vloží hodnoty zadané v properties souboru. Naopak při zavolání metody *getMappedProperties* se z proměnných objektu vytvoří properties soubor. Takto lze jednoduše ukládat objekt do lidsky čitelného souboru.



Obrázek 10: Třídní diagram modulu pro zajištění základní funkcionality komunikačního programu

6.8 Modul pro zajištění základní funkcionality komunikačního programu

Tento modul zajišťuje spojení funkcionalit okolních modulů pro jejich snadnější přístup vůči modulu grafického uživatelského rozhraní. Navíc implementuje třídy typické pro funkci komunikačního programu. Třídní diagram je vyobrazen na *„Obrázek 10: Třídní diagram modulu pro zajištění základní funkcionality komunikačního programu“*.

Pro definici metod, které bude moci modul grafického uživatelského rozhraní využívat, je zde rozhraní *IImCore*. V jeho definici lze nalézt například metody pro přihlášení, odhlášení, získání informací o aktuálním uživateli, poslání zprávy, načtení konfigurace a podobně.

Třída *ImCore* implementující toto rozhraní tyto metody definuje. Obsahuje vazby na třídy a rozhraní z ostatních modulů, jako například *ICommunicator* z komunikačního modulu.

Třída *DbSynchronizer* zajišťuje synchronizaci databáze s ostatními klienty. Data získává díky vazbám na třídy *UsersMapper* a *ContactListMapper* z modulu pro objektově-relační mapování. Pro komunikaci využívá třídy implementující rozhraní *ICommunicator*.

Třída *DbSynchronizer* obsahuje metodu *synchronize*, jež zahájí proces synchronizace. Mějme v síti dva komunikační klienty, A a B. Způsob, jakým tato třída synchronizaci provádí, je následující:

1. Je dán podnět klientu A pro zahájení synchronizace databáze.
2. Komunikační klient A vypočte hash své databáze. Hash se skládá z identifikátorů uživatelů a časů poslední aktualizace v databázi. Tyto data jsou seřazeny a složeny do řetězce. Následně je z řetězce vypočten hash pomocí metody MD5.
3. Komunikační klient A odešle zprávu, jako příjemce je nastavena konstanta *Message.RECEIVER_ALL* pro odeslání jako broadcast zpráva a jako typ je nastaven *TYPE_MY_DATABASE_HASH*.
4. Klient B tuto zprávu přijme, vypočte hash své databáze a porovná vypočtenou hodnotu s hodnotou přijatou ve zprávě. Pokud se hodnoty shodují, mají shodné data v databázi a proces končí.
5. Pokud se hodnota neshoduje, v databázích klientů je rozdíl. Je odeslána zpráva klientu A. Typ zprávy je *TYPE_MY_DATABASE_INFO*. Do této zprávy jsou navíc přiloženy hodnoty, jež reprezentují id uživatelů a čas posledních aktualizací záznamů v databázi, týkajících se těchto uživatelů.
6. Klient A tuto zprávu ze seznamem přijme. Projde postupně všechny záznamy v databázi a porovnává jejich datum změny s informací přiloženou ve zprávě.
7. Pokud porovnávaný záznam v databázi má starší datum poslední změny než je ve zprávě, nebo pokud v databázi tento záznam úplně chybí, vyšle klient A zprávu. Typ zprávy je *TYPE_SEND_DATABASE_USER*, jako příjemce je nastaven klient B a do zprávy je vložen identifikační údaj záznamu, který chceme od klienta B přijmout.
8. Pokud porovnávaný záznam v databázi má novější datum poslední změny než ve zprávě, nebo pokud ve zprávě tento záznam úplně chybí, vyšle klient A zprávu. Typ zprávy je *TYPE_MY_DATABASE_USER*, jako příjemce je nastaven klient B a jako přiložený objekt je vložen objekt třídy *SecuredUser*, reprezentující tohoto chybějícího uživatele. Navíc jsou do zprávy vloženy záznamy ze seznamu přátel.
9. Pokud klient příjme zprávu typu *TYPE_SEND_DATABASE_USER*, odešle na adresu odesílatele zprávu se záznamem uživatele, jak je popsáno v předchozím bodě.
10. Pokud klient přijme zprávu typu *TYPE_MY_DATABASE_USER*, zkontroluje, zda přijatý záznam obsahuje novější data a pokud ano, uloží jej do databáze.

Tato technika zaručuje synchronizaci databáze. Nevýhoda je v nutnosti přenášet vyšší objem dat v případě, pokud je v síti mnoho komunikačních klientů. Jelikož je však komunikační klient určen do menších sítí, je tato nevýhoda zanedbatelná.

Třída *ThreadDbSynchronizer* obaluje třídu *DbSynchronizer* a zajišťuje, že synchronizace bude prováděna v samostatném vlákně. Díky tomu může probíhat asynchronně a nebude brzdit ostatní procesy.

Tento modul obsahuje filtry pro přidání vlastností komunikačnímu klientovi.

Filtr *CryptHistoryMessageFilter*, implementující rozhraní *IMapperFilter,* je filtr vkládaný do třídy *HistoryMessageMapper* modulu pro objektově-relační mapování. Úkol tohoto filtru je šifrovat zprávy předtím, než jsou uloženy do databáze a dešifrovat je ihned po načtení. K tomuto účelu využívá třídu implementující rozhraní *IPasswordCrypt*.

Další filtr, *CryptMessageFilter*, implementuje rozhraní *IMessageFilter*. Vkládá se do třídy implementující rozhraní *ICommunicator* a slouží k šifrování přenášených zpráv. Šifrují se pouze zprávy typu *TYPE_MESSAGE*, jelikož ostatní zprávy zajišťují vnitřní chod komunikačního klienta.

Poslední filtr, *RecordHistoryMessageFilter*, implementuje také rozhraní *IMessageFilter* a zajišťuje ukládání přenášené komunikace.

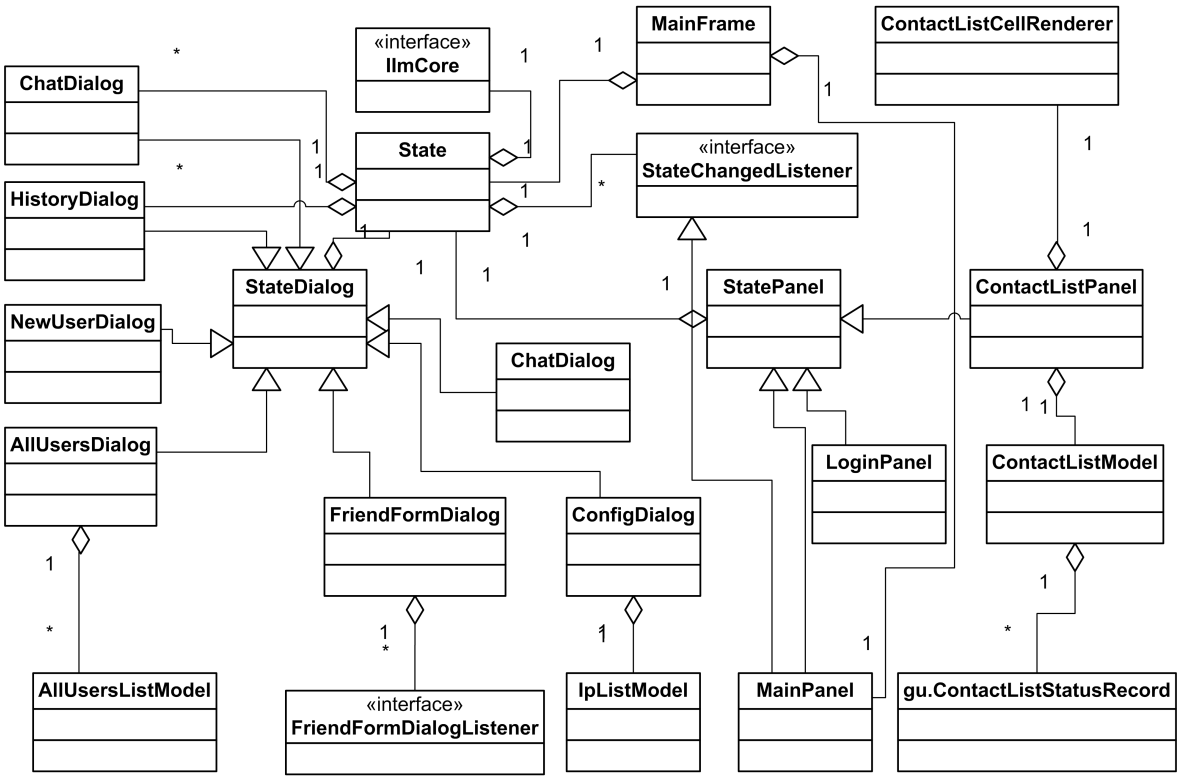
Dalším článkem modulu pro zajištění základní funkcionality komunikačního programu je množina objektů starající se o aktuální stav seznamu přátel. Centrální třídou pro zajištění této služby je třída *ContactList*.

Třída *ContactList* spravuje záznamy přátel, obsahuje metody pro jejich seřazení, přidávání, odebírání a podobné. V případě změny jejich stavu dokáže delegovat zprávu o změně stavu kontaktu. Tuto zprávu předává všem zaregistrovaným posluchačům implementujícím rozhraní *ContactListChangedListener*.

Třída *ContactList* obsahuje list instancí třídy *ContactListStatusRecord*. Tato obalová třída obsahuje instanci třídy *ContactListRecord* a další informace ohledně stavu daného uživatele. Hlavní dvě proměnné určující stav třídy jsou *lastSeen* a *online*. První jmenovaná obsahuje čas, kdy naposledy dal daný kontakt o sobě vědět, druhá pak výsledný stav.

Další třída, kterou *ContactList* obsahuje, je třída *ContactListDaemon, j*ež má spravovat záznamy ve třídě *ContactList* tak, aby byly vždy aktuální. Tato třída v definovaných intervalech vysílá zprávy všem klientům, čili nastaví parametr *receiver* na *Message.RECEIVER_ALL*, a jako typ nastaví *TYPE_I_AM_ONLINE_NOTIFICATION*. Díky těmto zprávám ostatní klienti ví, že uživatel je přihlášen. V případě, že klient nepřijme tuto zprávu po dobu delší, než je trojnásobek intervalu vysílání, usoudí, že klient se náhle odhlásil.

Pokud klient ukončuje svou činnost, automaticky se odešle zpráva typu *TYPE_LOGGING_OFF*, která naznačuje, že klient se odhlašuje, čili končí svou činnost.



Obrázek 11: Třídní diagram modulu pro grafické zobrazení komunikačního klienta

6.9 Modul pro grafické zobrazení komunikačního klienta

Tento modul má za úkol zobrazovat interaktivní prostředí, se kterým bude uživatel pracovat. Třídní diagram lze nalézt na *„Obrázek 11: Třídní diagram modulu pro grafické zobrazení komunikačního klienta“*.

Třída *State* zde reprezentuje stav grafického rozhraní, ve kterém se komunikační klient nachází. Obsahuje kromě objektu implementujícího rozhraní *IImCore* také seznamy otevřených komunikačních dialogů a dialogů s historií.

Instanci třídy *State* obsahují dvě abstraktní třídy, *StatePanel* a *StateDialog*. Tyto třídy jsou předky pro dialogy a okna komunikačního klienta. Díky této vazbě na třídu State mohou dialogy a okna dědící z těchto tříd získávat potřebné informace pro autonomní zobrazení.

Třída *MainFrame* zde reprezentuje vstupní bod programu pro běh aplikace. Tato třída zpočátku vytvoří instanci třídy *ImCore* a *State,* dále pak samotnou instanci třídy *MainFrame* a předá jí řízení. Třída *MainFrame* obsahuje třídu *MainPanel*, jež se stará o zobrazování správného panelu.

Třída *MainPanel* implementuje rozhraní *StateChangedListener*, díky němuž po registraci do objektu třídy *State* získává informace o změně stavu stránek. V případě podnětu pak zobrazí požadovanou stránku.

Po inicializaci programu se zobrazí instance třídy *LoginPanel*. Tento panel má za úlohu vyžádat od uživatele přihlašovací údaje a v případě přihlášení vložit hodnoty do třídy *State*. Z tohoto panelu lze také vyvolat konfigurační panel či panel pro vytvoření nového uživatele.

Zobrazení konfiguračního dialogu má na starost třída *ConfigDialog*. Tento dialog zatím obsahuje jediné nastavení, nastavení IP adresy. Na výběr jsou IPv4 adresy přítomné na počítači. Lokální adresy nejsou zobrazovány. Seznam položek v tomto dialogu zajišťuje třída *IpListModel*. Tato třída, *IpListModel* získává IP adresy ze základní třídy jazyka java, *java.net.NetworkInterface*. Pomocí metody *getNetworkInterfaces* jsou nalezeny v počítači síťová rozhraní a pak vyhledány a vyfiltrovány požadované adresy.

Pro založení nového uživatele je zde třída *NewUserDialog*. Tato třída slouží jako formulář, který se používá jak pro vytváření, tak pro editaci uživatele. Při každém použití je provedena validace, zda nejsou vyplněny invalidní hodnoty.

Po přihlášení uživatele je zobrazen panel, který obstarává třída *ContactListPanel*. V tomto panelu lze provádět příkazy pro editaci údajů, otevření okna pro konverzaci, otevření okna pro přidání uživatele do seznamu přátel a podobně. Tento panel implementuje rozhraní *MessageListener* a díky tomu přijímá zprávy od ostatních komunikačních klientů. Tento panel si filtruje pouze zprávy typu *TYPE_MESSAGE* a zobrazí obsah zprávy v konverzačním okně. Seznam konverzačních oken se nalézá ve třídě *State*. V případě, že konverzační okno s daným uživatelem ještě nebylo otevřeno, třída *ContactListPanel* toto okno vytvoří a zprávu vloží. Pro správu kontaktů pro třídu *ContactListPanel* je zde třída *ContactListModel*. Tato třída spravuje a formátuje data. O styl zobrazení se stará třída *ContactListCellRenderer*. V případě, že by se měl měnit vzhled seznamu uživatelů v okně, je nutné upravit právě tuto třídu.

O zobrazení konverzačního okna se stará třída *ChatDialog*. Tato třída implementuje rozhraní *MessageTimeoutListener*. Pokud se tedy nepodaří zpráva doručit, třída na tuto událost zareaguje a zobrazí informaci o nedoručené zprávě.

Z konverzačního okna lze zobrazit historii konverzace. K tomu slouží třída *HistoryDialog*. Tato třída načte všechny záznamy konverzace z databáze a zobrazí je v dialogovém okně.

Pro přidání uživatele slouží třída *AllUsersDialog*, ve které se zobrazí všechny dostupné kontakty z databáze, které dosud nejsou v seznamu přátel.

Po vybrání uživatele k přidání se zobrazí dialog, jež spravuje třída *FriendFormDialog*. V této třídě lze nastavit přezdívku uživatele, pod kterou se má zobrazovat, a skupinu, do které má patřit.

7 Popis aktuálního stavu vývoje aplikace

Komunikační klient, založený na předchozí analýze, je k nalezení na přiloženém CD. Jsou zde k dispozici jak zdrojové kódy, tak zkompilovaný komunikační klient, připravený k práci.

Dle zadání implementace obsahuje většinu vlastností. Bohužel, přenos souborů již z časových důvodů nebylo možno implementovat a tato vlastnost je zařazena jako další cíl vývoje. Také oznámení o změně stavu uživatele je implementováno pouze jako změna pozadí v seznamu uživatelů. O případném rozšíření, například přidání vyskakovacího okénka, se rozhoduje. V případě nalezení dostatečně sofistikovaného a pro uživatele ne příliš obtěžujícího řešení je možná implementace.

Systém je dostatečně modulární. Je připraven na případné úpravy v návrhu a změny funkcionality. Jako u každá aplikace je nutné dlouhodobější testování pro nalezení chyb v programu.

Dosavadní testování probíhalo na počítačích s těmito konfiguracemi:

Tabulka 1: Tabulka testovacích konfigurací

Procesor	RAM	OS	Java
Intel Core2 Duo	4GB	Windows 7 Professional 64bit	Java 6.0 Update 18 Sun
Intel Core2 Duo	4GB	Windows 7 Professional 64bit	Java 6.0 Update 16 Sun 64bit
AMD Sempron	1GB	Windows XP Professional SP3	Java 6.0 Update 18 Sun
Intel Pentium M	1,5GB	Windows XP Professional SP3	Java 6.0 Update 18 Sun
Virtuální systém		Ubuntu 9.10 "Karmic Koala"	Java 6.0 Update 15 Sun
Intel Pentium 4	1GB	Windows XP Professional SP3	Java 6.0 Update 7 Sun

V případě použití operačního systému Microsoft Windows komunikační klient funguje bez problému. Bohužel, při běhu komunikačního klienta v systému Ubuntu jsem nalez závadu při příjímání broadcast zpráv, jež znemožňuje jeho funkčnost. Na odstranění této závady jsem již vynaložil mnoho úsilí, zatím marně. Pro lepší použitelnost programu však bude nutné tuto chybu odstranit, a proto se jim v budoucnu budu zabývat.

Pro nalezení a odstranění dalších chyb či negativních vlastností komunikačního klienta je nutné delší testování.

8 Závěr

V předchozích kapitolách byla nastíněna analýza a funkcionalita vyvíjené aplikace (komunikačního programu) založené na modelu peer-to-peer sítě. Byly využity sofistikované metody pro snadnější, rychlejší a efektivnější vývoj této aplikace. Dále byla navržena možná řešení nalezených problémů, vyzdvižena jejich pozitiva i negativa a následně pak vybrána ta nejpřijatelnější pro implementaci.

Během vývoje se vyskytly problémy, a to jak během návrhu, tak při samotné implementaci. Ku příkladu určení identity uživatele v peer-to-peer síti, synchronizace databáze... Zjištěné problémy byly mnohdy ne příliš obvyklé a vyžadovaly samostatná řešení. Po analýze následovala implementace. Poznatky z analýzy byly shrnuty a zformovány do implementace programu. Pro odstranění nežádoucích chyb byl vzniklý program testován v různých prostředích.

V současné době je komunikační klient plně funkční. Vytvořený software je však ještě relativně mladý a je velice pravděpodobné, že během následujícího používání nastanou stavy v programu, které nebyly předpokládány a budou nalezeny chyby, které bude nutno odstranit.

Následný vývoj se zaměří na dokončení implementace chybějících vlastností, na testování, odstraňování chyb a přidávání funkcionalit pro snadnější práci.

9 Použitá literatura

[1] – BLACK, Uyless. *OSI : a model for computer communications standards.* Englewood Cliffs : Prentice Hall, 1991. 528 s. ISBN 0-13-637133-7

[2] – Oracle. Jaxb: JAXB Reference Implementation [online]. 2010 [cit. 2010-04-19]. *Jaxb: Unofficial JAXB Guide*. Dostupné z WWW: <https://jaxb.dev.java.net/guide/>.

[3] – SQLite Home Page [online]. 2010 [cit. 2010-04-19]. *SQLite Documentation*. Dostupné z WWW: <http://sqlite.org/docs.html>.

[4] – RAWSHAW, David. David Crawshaw [online]. 2010 [cit. 2010-04-19]. SQLiteJDBC. Dostupné z WWW: <http://www.zentus.com/sqlitejdbc/>.

[5] – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 31.7: The RSA public-key cryptosystem, pp.881–887

Seznam ilustrací

Seznam tabulek